A LIGHTWEIGHT DSP FRAMEWORK FOR OMAP3530-DRIVEN EMBEDDED DEVICES

BY

WENJIA ZHOU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor Geir E. Dullerud

ABSTRACT

This thesis provides a lightweight framework, called MiniDSP, for OMAP3530 heterogeneous dual core SoC to run tasks on its DSP co-processor. This framework is composed of a minimal DSP kernel and a set of programs which run on the ARM A8 master processor. The minimal kernel maintains system stability and initializes the interrupt handler. The set of programs includes a DSP device driver, a host program and two utility programs. Through the device driver, the ARM core can send commands to the DSP and control it to execute compute-intensive applications. The host program performs task off-loading and general ARM-DSP communication. Finally the two utility programs are responsible for converting the DSP executable to a bootable format used by the framework. This framework is open source, highly configurable and lightweight, enabling the possibility of high performance computing on DSP.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank all the lab members for their generous support: Seungho Lee, for patiently guide me through all the technical details of Hovercraft; Bicheng Zhang, for keeping the lab servers up and live; Steve Granda, for the help of debugging the system; Richard Otap, for all Gumstix software introduction; and Rohan Khanna, for helping me put the hardware pieces together. Finally, I would like to thank my adviser, Geir Dullerud, for advice and support on this project, and the opportunity to work in the lab. Without them, this project would not be possible.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

For the past few years, open source hardware has been a hot topic among different groups of people. With a large active online community just like open source software, people with little programming background and limited analog/digital circuit knowledge can do innovative projects using these hardware platforms. Arduino is one of the most easily and widely used platforms. However, a platform like Arduino has very limited computing power. More advanced developers have started making pocket-computers by utilizing ARM architecture and SoC technology. Beagle Board and Gumstix are two examples. They are equipped with TI OMAP3530, which is powerful enough to run Linux. This feature is very useful for fast prototyping; instead of making a customized board, we can just plug and play. In order to keep pace with fast development of the hardware, more software solutions to adapting multiple cores have been developed. For example, the C6EZ series of products published by Texas Instruments is a platform enabling easy development on ARM+DSP board. In this thesis, we provide an open source framework called "MiniDSP Framework" for offloading workload between different cores on OMAP3530 processor inside Gumstix pocket-computer and an application which utilizes this framework on Hovercraft platform.

The thesis is organized as follows. Chapter 1 is a brief review of the Hovercraft platform and why we started developing this framework. Chapter 2 explains the design of the framework. Chapter 3 is a profiling test, and Chapter 4 is the application which uses this framework. In Chapter 5, we discuss the future work that will improve the framework and the application.

1.1 Review of Hovercraft

1.1.1 Robot Hardware

Hovercraft has a light foam body. It has five ducted fan thrusters, four of which are used to control the moving direction and acceleration, while the other is used to generate lift so that it can slightly float above the ground. The Hovercraft runs indoors with a six-camera overhead vision system. And as mentioned previously, it has a main on-board controller which is Gumstix and a second F28335 DSP board for thruster control. Fig 1.1 is a photo of the Hovercraft main board.

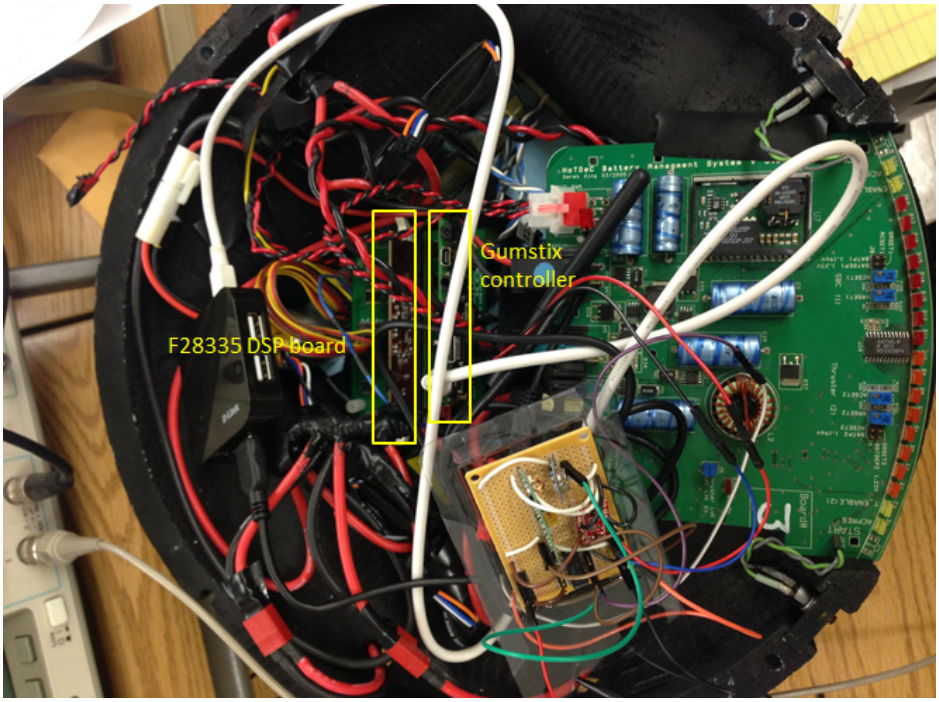

Figure 1.1: Hovercraft main board

1.1.2 HoTDeC System Overview

Hovercraft is mostly used for control algorithm experiments. Six overhead cameras capture photos and send them to Vision Server via USB. The Vision Server combines them into one calibrated image and then runs a detection algorithm to track the positions of all Hovercrafts in the course. The Vision

server also broadcasts the position through the local wireless network to all clients in the network.

The Hovercraft control algorithm is running on another machine in the network. Usually, there are different control algorithms to make the robot run autonomously according to certain game rules. Another host machine takes the position data and computes the next position the Hovercraft should move to. The new location is sent to Gumstix via WiFi. Gumstix will then move the Hovercraft to the desired location.

1.1.3 Gumstix Board

Gumstix is an open source hardware platform. All schematic designs and layouts are released on the company's website. It comes with OMAP3530 heterogeneous dual core OMAP3530 processor, which is consisted of an ARM A8 core as MPU and a high-performance Image, Video, Audio (IVA2.2) accelerator subsystem. Our framework is developed for this OMAP3530 processor. In addition, the Gumstix board has integrated 802.11b/g WiFi, Bluetooth, power management chip and external memory.

Gumstix is the main controller for Hovercraft. It runs Linux and handles wireless communication with other components in the HoTDeC system, such as the Vision Server. But Gumstix does not control the thrusters directly. It talks to the F28335 DSP board via serial connection and passes thruster control parameters to the DSP. F28335 then uses a PID controller to generate a PWM signal and adjusts the thrusters.

1.2 Motivation

The Hovercraft in the HoTDeC project uses Gumstix computer as the primary controller. Gumstix's processor OMAP3530's MPU, ARM A8, has a ported Linux build. However, this Linux build does not support the use of OMAP3530 DSP subsystem, which means the DSP subsystem is wasted. A separate DSP board F28335 is used to control the six motors on the Hovercraft as we mentioned previously. In fact, OMAP3530 itself has 6 PWM pins which are enough to control all the Hovercraft thrusters. If we can take advantage of the DSP inside the OMAP3530 processor, the F28335 separate

DSP board can be removed. Moreover, as robots are doing more and more computing intensive jobs such as media signal processing, the power of DSP is helpful.

1.3 Related Work

1.3.1 DSP Background Review

A DSP chip is basically a microprocessor whose architecture is optimized for performing typical DSP operations on sampled data at a high rate [1]. ARM, or any other general purpose processor, can also perform these computations, but is not specially designed to do so. The DSP core in OMAP3530 can execute 8 instructions/cycle (it has 8 execution units) and provide optimized instruction set for video and image processing [2].

1.3.2 DSP Gateway

The DSP Gateway consists of two parts: a Linux device driver on the ARM and a DSP-side kernel library, and they communicate with each other. The Linux part provides conventional Linux device driver interface so the application can use DSP functions through normal system calls such as read() and write(). The implementation uses mailbox mechanism and inter-processor buffer to enable the communication between ARM and DSP core. Then, by setting up the Gateway device driver in Linux, the inter-processor communication can be successfully achieved [3]. The DSP kernel library part provides a multi-task environment and APIs for user tasks, which can be controlled from Linux (Fig 1.2). This is one of the earliest works in the field developed by Nokia for OMAP1 and OMAP2 but not fully tested with OMAP3. And the development has halted since 2010 [4].

1.3.3 DSP Bridge

The DSP bridge is developed by TI. The DSP bridge driver provides features to control and communicate with the DSP, enabling parallel processing for

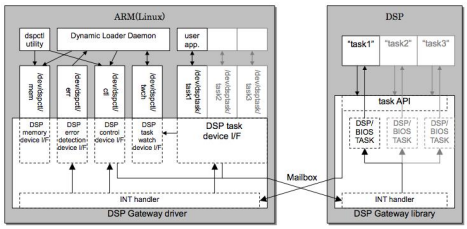

Figure 1.2: DSP Gateway components layout

multimedia acceleration [5]. It enables the applications running on MPU to offload the processing to DSP. Similar to the DSP Gateway mentioned above, the GPP program normally treats DSP core as a device in the system. Since the DSP provides service real-time task, it sets up a data stream (device independent) between the GPP interface and DSP interface on the application level while link driver (equipped on both cores) sets up low level communication between the processors. Instead of serving as a single device in Kernel on ARM, DSP core runs its own system RTOS (Fig 1.2) [6].

1.3.4 DSPLink/SYSLink

DSP/BIOS LINK is foundation software for inter-processor communication across the GPP-DSP boundary (and luckily it is a product currently being maintained). The library provides a generic API that abstracts the characteristics of the physical link connecting GPP and DSP from the applications. It eliminates the need for customers to develop such a link from scratch and allows them to focus more on application development. SysLink is composed by the process manager, inter-processor communication mechanism and utility modules. It has no constraint on the system run on a different processor (it can be either a higher level operating system or as simple as RTOS, like SYS/BIOS). See Fig 1.3 and Fig 1.4 for the module layout of SysLink architecture in both operating systems [7].

As the name suggests, DSP/BIOS is expected to be running on the DSP. No specific operating system is mandated to be running on the GPP. It is released on a reference platform for a set of reference operating systems. The release package contains full source code to enable the customers to port it to their specific platforms and/or operating systems. We want to figure out how to build this DSPLink as part of our project. SysLink has a few important

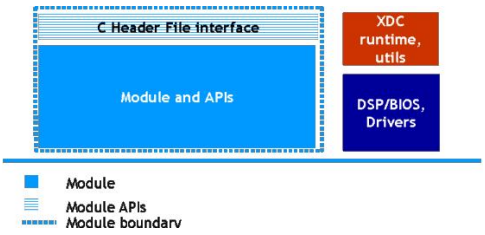

Figure 1.3: DSPLink module layout Sys/BIOS

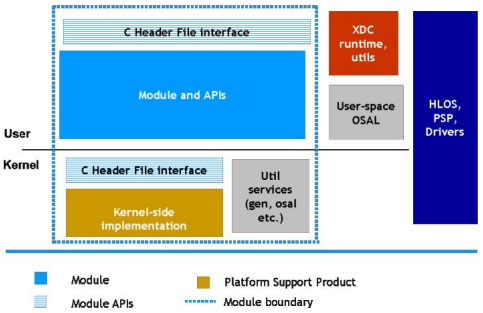

Figure 1.4: Higher level operating system

mechanisms we need to focus on [7]:

- System Manager: Provides full control to all modules in the system, including initialization and memory allocation, configuration and finalization.
- Processor Manager: Provides processor port for each processor and also provides power manager interface.
- Inter-Processor Communication Protocol: Enables lightweight communication between processors. This is very important because the inefficiency between the processor communication can generate overhead which slows down multicore processing and possibly generates lower computing power than its single core competitor.

1.3.5 C6EZRUN

The C6EZRun project allows you to seamlessly use the DSP from the ARM core on TI's ARM+DSP devices, without having to deal with any advanced, and potentially complicated, frameworks and software stacks. C6EZRun is a set of tools which will take in C files and generate either an ARM executable,

or an ARM library which will leverage the DSP to execute the C code [8]. This project provides a feature to extract certain critical functions out and form a library which runs on DSP. So when the user calls the critical function, it is actually executed on the DSP. It is built on top of DSPLink, and is not covered by our project.

1.3.6 DVSDK

The OMAP35x Linux DVSDK is a free non-commercial Linux that comes complete with everything a user needs to get started quickly and can be shipped in production systems. The Quick Start Guide should allow users to connect the hardware and start playing with GUI based demos which demonstrate the hardware capabilities in a matter of minutes. Furthermore, developers should be able to install all the software (includes source code) and development tools in less than an hour and be on their way to development [9].

Even though this is a complete software package, since most people have an existing OS and have many software packages based on that particular kernel, it is not a good idea to abandon what they currently have and install the whole DVSDK on the Gumstix Overo.

1.4 About Related Work

Although the existing projects seem to provide a complete and feature-rich software package for the communication between GPP and DSP, installing them is very complicated and platform dependent. But in the cases of most open hardware users, the DSP usage they require, compared to the functionality these package provided, is very primitive. So a lightweight DSP framework will fill this vacancy.

CHAPTER 2

DESIGN

As we described in the previous chapter, the problems of the current DSP-bridgeware are the following: first, the majority of open source software is outdated; second, TI's official software is overkill for most of the amateurs who work on Gumstix and Beagleboard; third, no open source solution is independent from TI's closed source DSP/BIOS RTOS. Thus, a new DSP-ARM bridge should meet the following requirements: first it should be lightweight and fulfill the basic needs; second, it should be simple to use compared to existing solutions; last it should be highly configurable and not bounded to any closed source code. MiniDSP framework meets all the requirements.

2.1 Framework Architecture Overview

In MiniDSP framework, DSP is treated as a passive stand-alone co-processor. ARM works as a master. Two cores use Mailbox Module and a shared circular buffer to communicate, shown in Fig 2.1.

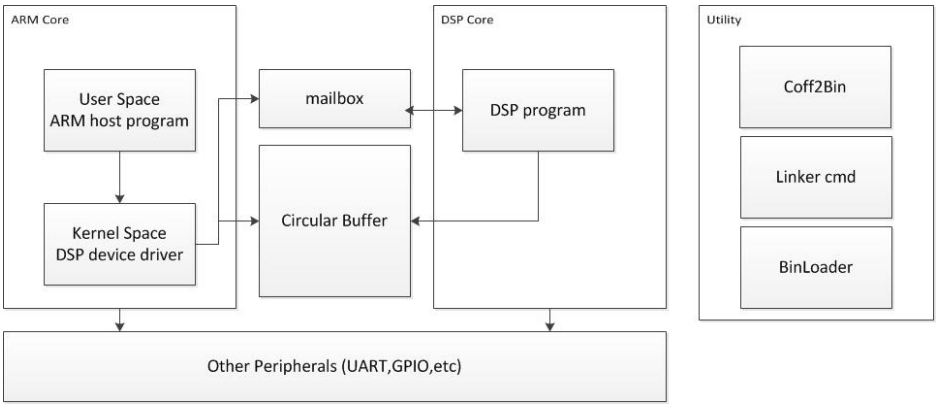

Figure 2.1: Architecture

ARM core and DSP core refer to the MPU subsystem and IVA2.2 subsystem, respectively (Fig 2.2). Since Linux is running on ARM core, DSP is treated as an external device under Linux. We need a device driver to control DSP hardware. A host program in Linux User Space talks to the device driver. Device driver interacts with mailbox and shared buffer. DSP also reads/writes data to them. Mailbox and shared buffer are the major methods by which two cores communicate. Peripherals can be either owned by ARM or DSP. In Fig 2.1, the utility part is also very essential to the framework. In brief, the Linker command file is used to compile the DSP program and generate COFF executable. Then Coff2Bin will extract binary information out from COFF executable. Finally, the BinLoader will take the output from Coff2Bin and load it into DSP physical memory. Each component will be explained in detail in later sections.

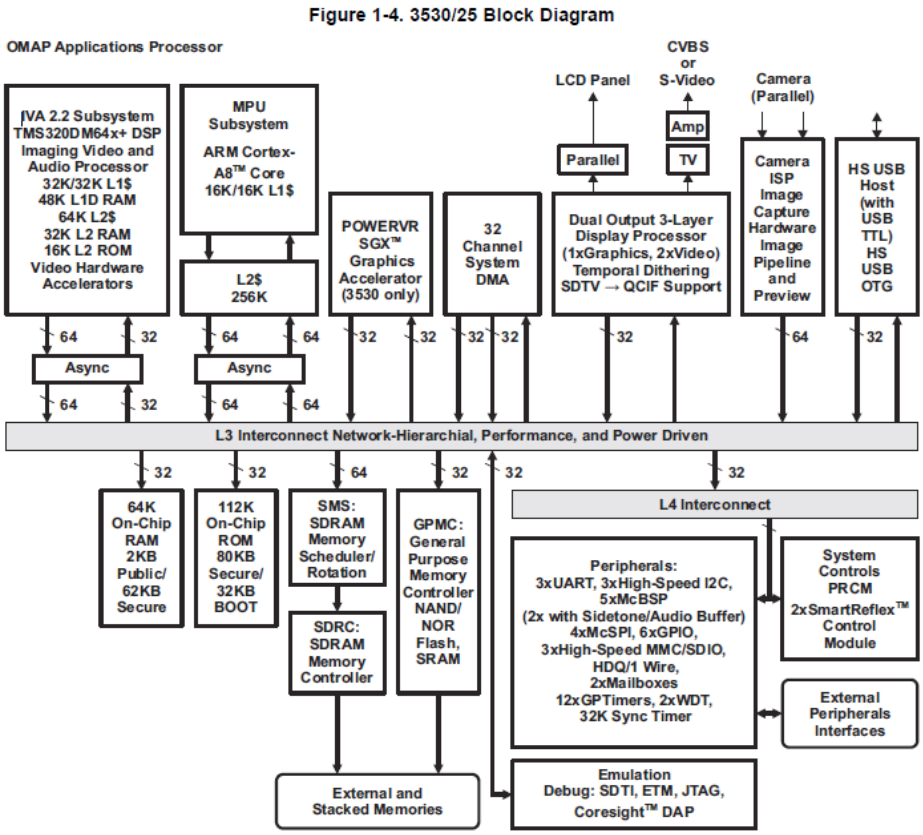

Figure 2.2: OMAP3530 block diagram

2.2 DSP Device Driver

A char device driver is implemented to support DSP hardware access under Linux. Several steps need to be performed inside the driver.

The first step is IO address mapping. Usually hardware module registers are memory mapped, but their addresses are not visible to Linux, so they have to be mapped to kernel virtual addresses before the kernel can access them. Second, mailbox module should be initialized in the driver as this is the only way for inter-processor communication. Sending and receiving mails can be configured to either polling or interrupt mode. The framework uses an interrupt based method because it is more efficient. Kernel driver is responsible for registering the IRQ for the mailbox module. IRQ number for the mailbox module is given in the reference manual [2].

Users may want to use other modules as well. These modules can be initialized in programs running on either core. A module like mailbox is always used in framework no matter what DSP program it is, so it is better to put mailbox initialization inside the driver to avoid duplicated code in all user programs. Modules like UART which are used case by case can be set up in user programs to reduce kernel driver size.

At boot time, 32 MB memory is reserved from Linux. The layout is show in Fig 2.3:

Figure 2.3: Memory layout

The RAM area that is dedicated to DSP programs starts at 0x8E003000 (configurable). The starting address of the compiled DSP program should be equal to or larger than this, and should be 1k aligned [2]. This part of the RAM can be memory mapped to user space, allowing the host program

on ARM to efficiently load new programs onto the DSP. If it is not mapped and the host program wants to read/write to the device memory address, it has to call the device driver ioctl function. This can introduce a lot of overhead, especially if there are a lot of read/write operations. Memory mapping is the only way to transfer data between user and kernel spaces that does not involve explicit copying, and is the fastest way to handle large amounts of data, so implementing mmap can speed up loading time for the DSP program.

2.2.1 DSP Drive APIs

DSP is a slave processor, so the driver should provide full support to reset/release the DSP, as well as general read write register ability. The current driver has the following ioctls:

```
DSPDD_IOC_RESET:
Perform a hard reset on DSP core
DSPDD_IOC_BOOT:
Take DSP program starting address as parameter and
release DSP from reset.
DSPDD_IOC_READREG:
Read value from the register of the given address.
DSPDD_IOC_WRITEREG:
Write value to the register of the given address.
DSPDD_IOC_MSG_READ:
Blocking call to read 1 message (4 bytes) from mailbox.
DSPDD_IOC_MSG_WRITE:
Send 1 message to DSP mailbox, immediate return.
```

The driver also has:

```
dsp_read: ssize_t dsp_read(int fd, void *buf, size_t count)
```

The behavior of this function will read up to *count* of bytes. If there is less than *count* new data available in the circular buffer, return all the unread bytes. The circular buffer has a read pointer and a write pointer. Assume

DSP is the data producer and ARM is the consumer. Every time DSP has new data written into the shared buffer, the write pointer is incremented. And every time ARM reads from the shared buffer, the read pointer is incremented. If the write pointer is larger than the read pointer, it means there is unconsumed data for ARM. In some cases, the buffer is wrapped around, so the write pointer can be less than the read pointer and the buffer still have unconsumed data. ARM issues a read call with *count*; if the amount of unread data is less than *count*, the amount of unread data is returned. If *count* is no larger than unread data, only *count* bytes are returned.

2.3 DSP Minimum Kernel

When the host program releases the DSP from sleep, based on the value in the IVA2_BOOTMODE register, the DSP will start to execute the instructions which pointed by the register IVA2_BOOTADDR or jump to a fixed address which configures bootloader. In this project, the former mode is used. The host program performs part of the bootloader job. When the DSP is released, the application is already loaded in the RAM, so there is no need to write another bootloader. But it also leads to a problem that when the application ends, the system crashes. This is caused by this direct jump to the starting address of the application. When the program ends, there is no return address that the PC can refer to, thus causing systems to hang.

The solution is to introduce a minimum kernel, start execution from the kernel and let the kernel jump to the first application. In this way, the register will have a valid return address stored. When the application finishes execution, PC will jump back to a known location. Since the DSP and ARM communicate using mailbox module, the DSP should be able to receive mailbox interrupt from ARM so that it can respond to ARM. In the minimum kernel, we should set up the interrupt vector table properly. The user can register his own interrupt service routine in the DSP user program, but also must check manual [2] to map the mailbox event to the correct interrupt.

In order to implement a framework which supports multitasking, the plan is to make mailbox interrupt work similar to INT 80. So when the ARM host program sends an interrupt to the DSP, with an address as the mailbox message, the DSP should be able to jump to that address and start execution

again.

2.4 ARM Host Program

A host program is required to help the DSP initialize and load programs to memory. An example host program for general debug and development is provided. A prompt will ask for user input to

```
Reset DSP
Release DSP
Quit host program
Run a program
Read register value
Write register value
Read message from mailbox
```

To execute a program, the user needs to select command 4 and provide the name of the boot table generated from Coff2Bin in Section 2.5.2. Then this host program will call the BinLoader in Section 2.5.3 and load the binary boot table into DSP memory. Then the program will be executed once the user releases the DSP from reset state.

2.5 Utility

2.5.1 Linker Command File

The linker command file tells the compiler how to compute a physical address that matches the hardware settings and the memory space configuration when the program executes [10]. For example, we specify the size of heap and stack, and where the stack pointer is in the linker command file. Refering to Fig 2.3, circular buffer occupies 0x8E00 0000 to 0x8E00 0FFF. No executable code should appear in that address range, so the DSP compiler should not even be aware of that address range. The DSP program has to be loaded in RAM section. In boot.asm, there is an interrupt vector table and Kernel code. They have special requirements for where they should be loaded. The

interrupt vector table always sits on top of the memory (since the DSP does not see the circular buffer, the interrupt vector table is on top in this case). The Kernel does not have a restriction on where it sits, but to make life easier, a reserved section is used to specify the location of Kernel code. A linker command file consists of two major parts: MEMORY and SECTIONS.

```
MEMORY
{
VECS: o = 0x8E001000 l = 0x1000 /* Interrupt Vector Table */
KERN: o = 0x8E002000 l = 0x1000 /* Kernel */
RAM: o = 0x8E003000 l = 0x01FFD000 /* RAM for DSP Programs*/
}
```

The purpose of the MEMORY directive is to assign names to ranges of memory. These memory range names are used in the SECTIONS directive. Here, as the memory name indicates, VECS is reserved for the interrupt vector table, KERN is where a minimum DSP kernel lies and the RAM is for all the DSP applications to run. SECTIONS forms output sections from input sections, and allocates those output sections to memory.

```
SECTIONS
{
.vecs: palign(1024) > VECS
.kern : palign(1024) > KERN
boot > RAM
{
     -l  rts64plus.lib<boot.obj>(.text)
}
...
}
```

For example, manual [2] specified that bootaddress should be 1 KB aligned locations. In this case, we make .kern aligned with 1 KB and put it into KERN memory range. Violation of this rule can cause system hang. Boot is an output section formed by boot.obj input section. And boot section is placed in RAM memory range.

2.5.2 Coff2Bin

The executable format generated by TI compiler tool chain is COFF (common object file format) [11]. However, COFF cannot run directly on bare metal DSP core because it contains application code and initialized data (which is needed to run the DSP standalone) along with information for debugging and linking.

Usually a DSP development board bought directly from TI comes with a JTAG debugger, and Code Composer Studio will help flash the COFF onto the DSP chip. In that case, CCS takes care of all the steps presented in the Utility part2.1. But now, these processes need to be done manually.

To address this problem, one should extract the application code and data from the .out file into a simpler table, called the boot table and later use a host program to copy data into DSP. The COFF structure is in Fig 2.4.

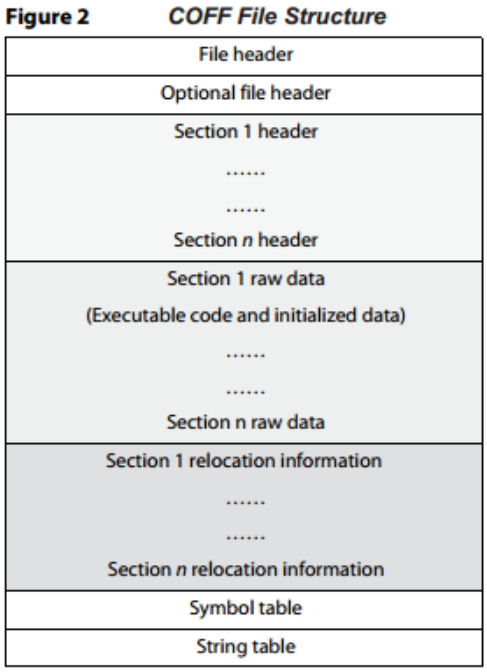

Figure 2.4: COFF file structure

- File header: This 22-byte-long header describes the general information of this object file.
- Section header: This header file defines where the section begins in the object file. To boot is to copy raw data into DSP memory, so there are

several important addresses in this section.

- Source Address: File pointer to raw data
- Destination Address: Load address of the section
- Byte Count: the Section size
- The flags (bits 40-43) determine if a section should be included in the boot table or not. Only text, data and vector table sections are needed for boot-up.
- String table contains extra information for the section name; its address is dependent on the size of the symbol table.
- Relocation information is empty most of the time.

The main idea of the Coff2Bin parser is to extract the file header and find out the number of sections the program contains, and then loop through all the sections to determine whether a particular section is needed for execution. If it is necessary, compute the address where raw data should be loaded, extract out the raw data from the section, and write to a binary file. A detailed flow chart is provided in Fig 2.5.

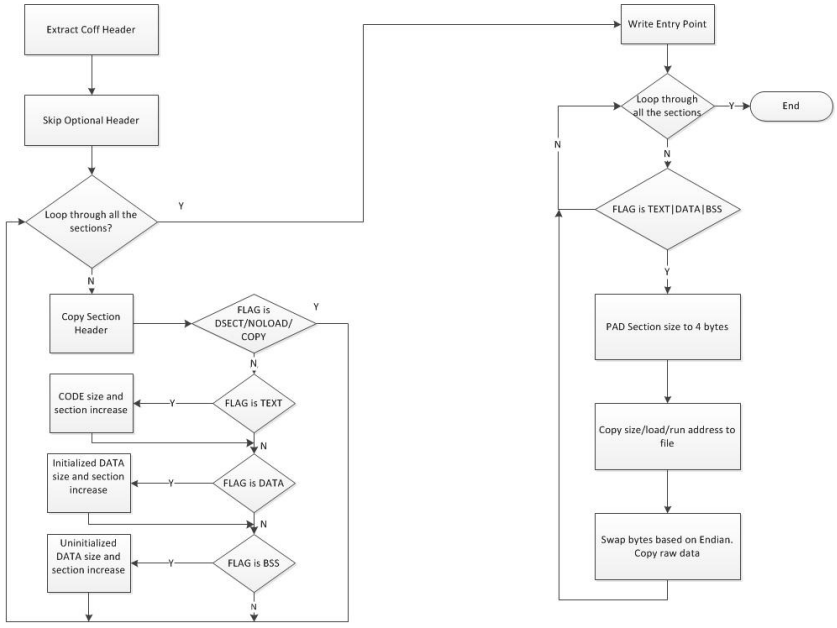

Figure 2.5: Parser flow chart

Table 2.1: Boot table

Entry Address
Section 1 size (4 bytes)
Section 1 load address (4 bytes)
Section 1 run address (4 bytes)
Section 1 raw data (4 n bytes)
Section 2 Size (4 bytes)
Section 2 load address (4 bytes)
Section 2 run address (4 bytes)
Section 2 raw data (4 n bytes)

2.5.3 BinLoader

When the DSP is released, PC of DSP will start from a given address and execute. BinLoader is responsible for providing the starting address to DSP PC and also writing application binary code to DSP memory. Coff2Bin provides a boot table for BinLoader, which will use the load address Table 2.1 and write raw data to the physical address [12]. An example boot table format is as follows: Based on this boot table, for each section, BinLoader will read the load address, skip the run address and copy all the raw data to memory. Finally, BinLoader will return the entry address. This entry address will be applied to Boot Address register in DSP.

CHAPTER 3

MESSAGE LOOP LATENCY TEST

3.1 Test Procedure

The message loop latency test measures the RTT of a 4-byte message sent from ARM to DSP and back. The test is split into two programs. One runs on ARM, and the other on DSP. ARM program generates an integer and calls ioctl to write the integer to the DSP mailbox. After the write, ARM waits for a blocking message read function call to return. The test program which runs on DSP keeps polling the mailbox status register; once it detects a value change, it will consume the message from ARM and put the same integer into the ARM mailbox. The write to ARM mailbox triggers an interrupt in Linux. In the mailbox interrupt handler, the condition that blocks ARM program will change. Eventually, the read function returns with the integer value sent from DSP. ARM program checks whether the value from the mailbox matches the one it sends out initially. If they do not match, the program will emit an error message showing the original value and the received value.

The reason for running this loop latency test is that we want to know how much overhead the MiniDSP framework can potentially introduce. The purpose of the framework is to utilize DSP computing power and balance load on the ARM core. One example is that the DSP gathers data and processes them without ARM's awareness. When DSP finishes processing, ARM will be informed of the result. From this example, if the framework introduces a large overhead, the task deadline may be missed. And it may be even faster to have the job scheduled natively on ARM. So the loopback latency is a major factor affecting the framework performance.

3.2 Test Result and Analysis

Table 3.1 is generated by running the test with different numbers of iterations. Each test with the same number of iterations runs three times. This can reduce the jitter from the Linux kernel scheduler. Fig 3.1 uses average of the three runs.

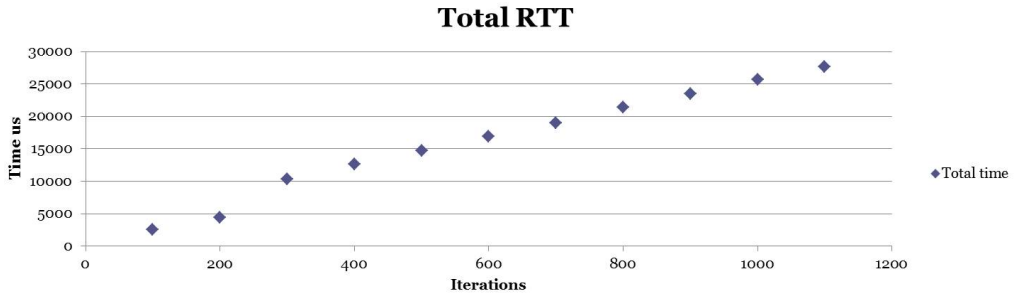

Figure 3.1: Round trip time plot

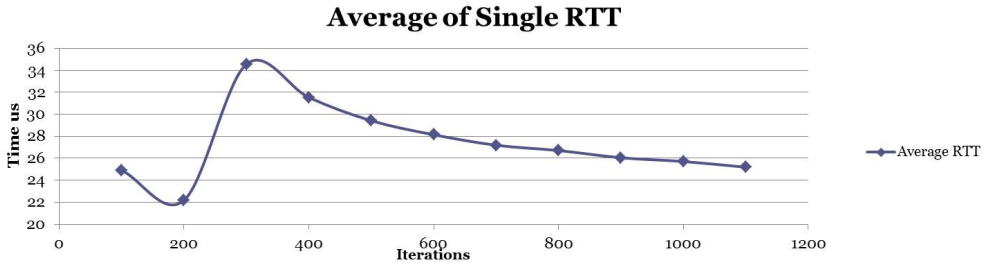

Figure 3.2: Single round trip time

The RTT is expected to increase linearly with the number of iterations, and Fig 3.1 confirms this. But Fig 3.2 indicates that single RTT of 300 iterations is much higher than single RTT of 200 iterations. After that, RTT gradually decreases to 25 μs. A possible cause is that the blocking call of the message read function waits on a variable, which is changed inside the interrupt handler. It is possible that after Linux serves the interrupt, another process is scheduled before the driver process. The longer the test runs, the lesser the influence of the scheduler, because the overhead upon single RTT becomes negligible.

The latency test only calculates the Mailbox overhead between IPCs. Depending on the application, the DSP and ARM may read and write data to shared memory. Then, we also need to include the time of memory read/write

Table 3.1: Loop Test Result (Result in μs)

Iterations	Run 1	Run 2	Run 3	Average time	Average RTT
100	2563	2624	2289	2492	25
200	4457	4425	4425	4436	22
300	10377	10254	10467	10366	34
400	12452	12451	12939	12614	31
500	14710	14618	14801	14710	29
600	16784	16969	16906	16886	28
700	19043	18920	19133	19032	27
800	22430	21393	20265	21363	27
900	23400	23407	23532	23446	26
1000	25637	25940	25542	25706	26
1100	27832	27560	27743	27712	25

when counting the overhead. Because shared memory is not cached, frequent ARM/DSP communication can hurt the performance. To address this issue, the user can configure the shared memory to use DSP L1SRAM or L2SRAM for better speed. If the user needs a large memory buffer to share data between two cores, external memory is still the first choice.

CHAPTER 4

DSP FRAMEWORK APPLICATION

Hovercraft wireless jamming experiment is the first application on this framework. It demonstrates that DSP is capable of helping ARM gather data. The experiment assumes that there are two teams playing a game, each consisting of two hovercrafts. Each team will emit a signal to jam the other team's communication while moving around to avoid being jammed.

4.1 Prototype Overview

The hovercraft has a 2.4 GHz wireless transceiver which is used to connect to the Networked Autonomous Vehicle Lab's local area network. But this frequency is less than ideal because of overcrowding and availability of jammers at this frequency. And Hovercrafts need wifi to communicate with the Vision Server to get their location; if the 2.4 GHz signal is jammed, the whole system will not function properly. In order to avoid this problem, 434 MHz and 315 MHz are chosen. So Team 1 deploys 434 MHz radio frequency transmitter and receiver pair for internal communication, and a 315 MHz transmitter to work as a jammer. For Team 2, 315 MHz is used as the major communication channel.

4.2 Design Model 1 Time Division Communication

The first design of the wireless communication module has the following components. One 434 MHz transmitter and receiver pair is the major channel for one team to stay connected. But the issue is that both team members cannot send out messages at the same time because they use the same radio frequency. In order to solve this, each team member will send out messages alternatively based on time division. An analog switch is used to fulfill this

need. One example is that team member A sends out messages only at even seconds, while team member B sends out messages at odd seconds.

The previous plan was to use a technique similar to wireless network carrier sensing. Under this technique, when team members want to send messages to each other, instead of sending out messages directly without considering conflict, each member will listen to the communication channel to detect if there is any incoming message. If there is, the member will sleep for a random length of time and try again. This plan was not carried out for several reasons. First, the RF modules used on the Hovercraft are very sensitive and always pick up noise in the air. It is hard to tell whether the received information is noise or useful information. Thus, it is not reliable. Second, as the game introduces a jammer, carrier sensing may mistake a jammer signal as a valid message. Thus, it is easier to force team members talk in turns.

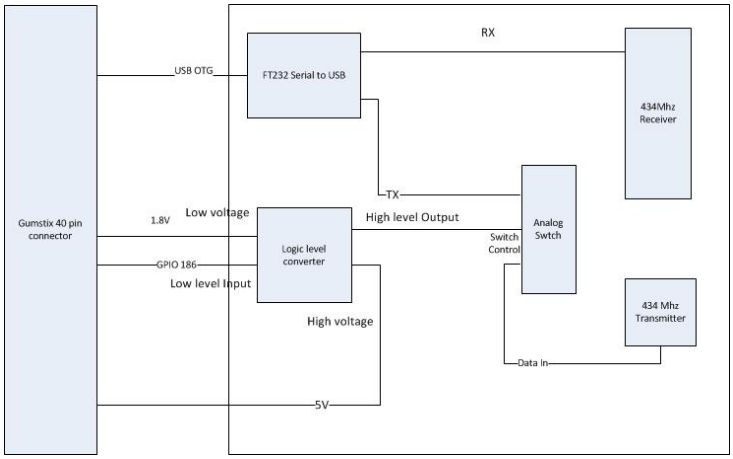

Figure 4.1: Design 1

Fig 4.1 is a high level block diagram of the hardware prototype to run the wireless jamming experiment. The analog switch connects the TX pin of the FT232 board and DataIn pin of the RF transmitter. The switch controller is a GPIO pin from Gumstix pinout. This GPIO output is used to control the Time Division communication. It outputs HIGH when TX is supposed to transmit signal. Since the Gumstix logic level is 1.8 V but the analog switch requires 5 V to drive, a logic level convertor is also needed.

The USB to Serial board in Fig 4.1 is used because at the time of the design, the pin heads were not accessible if the Gumstix was plugged into the Hovercraft main board, which it always was. Later, an extension connector

is soldered, so in the next design, the FT232 board is no longer used.

4.3 Design Model 1 Software

In Design Model 1, the program runs under Linux. It does not use the MiniDSP framework. ARM talks to the UART pins. The program is multithreaded. There are three threads: sending, receiving, and controlling. Because the RF module picks up a lot of noise, in order to increase the correct data receiving rate, a 4 byte header is inserted at the front. So the sender thread will pre-pend a 4 byte message when sending out the data. The receiver thread has to detect the header. It will keep reading the incoming data, finding the first matching byte of the header, and validating the rest. If the header is received correctly, the chance that the data is also correct will be much higher. The controlling thread is for time division. Since GPIO184 controls the analog switch, it will be toggled based on the system time. At even seconds, GPIO184 outputs low, and the analog switch will be disconnected, thus cutting of the input to the transmitter.

When one team member sends out a message, he also receives his own message. Under this situation, if the outgoing message does not match the incoming message, jamming is detected. For the other team member which does not send out any signal, it detects jamming by counting the valid data received in a certain amount of time. For example, if a member receives 1000 bytes without any valid header information, it can conclude that jamming is happening.

The disadvantage of the time division method is not scalable. Each Hovercraft will have to hard code its pre-configured time when it is allowed to send out messages to the team.

4.4 Design Model 2 Base Station

An essential part of the wireless jamming experiment is to model the signal strength of the transmitter, as well as the jammer. So if a team member detects that it is jammed, it will know the distance to move. To simplify the measurement, a new model is designed. In Design Model 1, each Hovercraft

can send out messages to the other one, but in Design Model 2, one team member acts as a base station which stays still at one location and keeps sending out signals. The other member does not send out any signal, but only receives messages from the base station and moves around to avoid the jammer.

This change reduces the number of hardware components used by the prototype board. To save some work, the same board from the previous design is rewired. The boards for the transmitter and receiver are different now. Refer to the base station block diagram in Fig 4.2; receiver Data Out is disconnected and analog switch is always closed. On the other hand, the receiver Hovercraft (Fig 4.3) shows the opposite connection.

The jammer circuit is very simple so the schematic is not shown here. It is a single 434 MHz or 315 MHz RF transmitter connected to an Arduino which is programmed to keep sending out random data. Fig 4.4 shows the circuit which provides 5 V to the RF module.

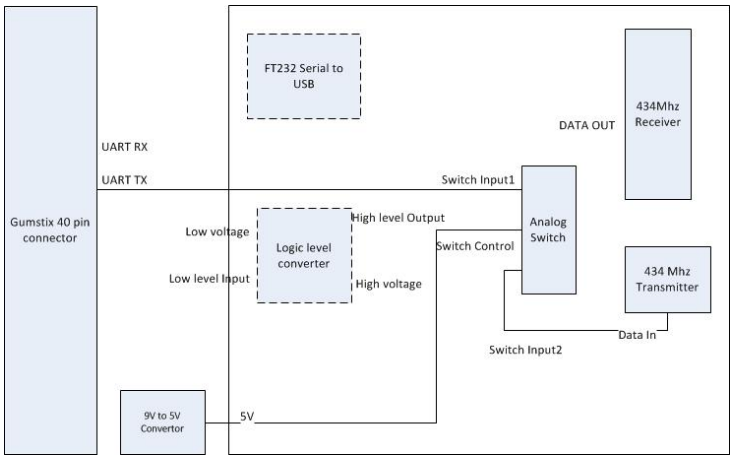

Figure 4.2: Transmitter

4.5 Design Model 2 Software

The software for Design Model 2 uses the Mini DSP framework. In contrast to the previous model, the DSP interacts with UART hardware. The software is separated as a host program (host-uart) which runs under Linux, and a DSP program. Host-uart uses the DSP device drive and keeps reading from the shared memory and prints out what is received to the standard output. For

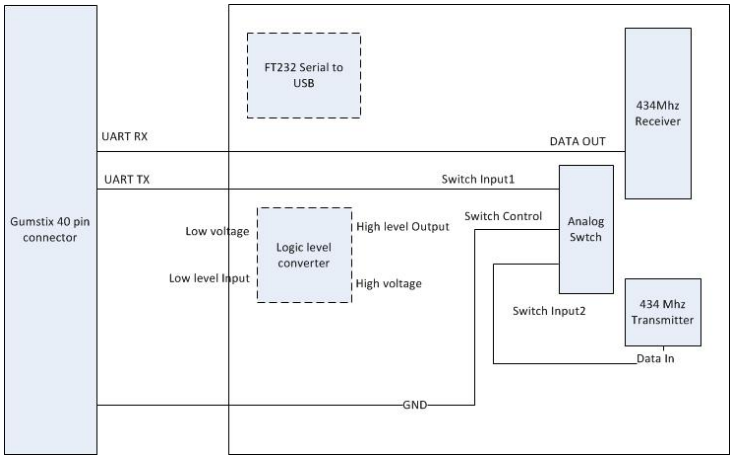

Figure 4.3: Receiver

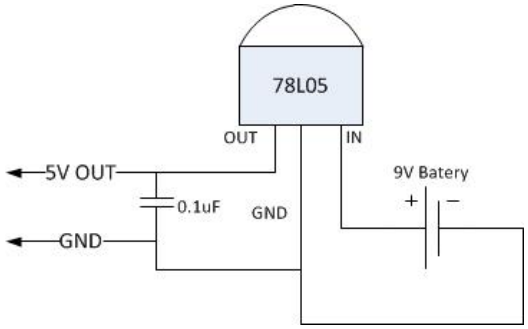

Figure 4.4: 9 V to 5 V convertor

the controlling DSP and loading program, it utilizes the debug host program which is described in section 2.4.

Since DSP owns the UART module, the initialization of the UART should be put into the DSP program. The other part of the program is similar to what is described in the previous section.

4.6 Wireless Jamming Experiment and Analysis

The experiment is to find the pattern of the jammer and base station signal spreading in the lab space, and determine the distance and direction a Hovercraft should move to get rid of the jammer.

The setup for the experiment is in Fig 4.5. For all runs, the base station is fixed at right with a star symbol. The jammer stays at the same place within each run, but varies between different runs. The test field is 9×10

tiles. This is the maximum area that this particular base station signal can reach and be stable, as well as visible to the overhead camera. In this field, 35 test points are picked. At each point, the status of the Hovercraft is either jammed or not jammed. A graph is generated based on this. The location of Hovercraft is reported by the overhead vision system. Some points along the boundary are a bit off due to camera distortion, but it does not affect the testing result.

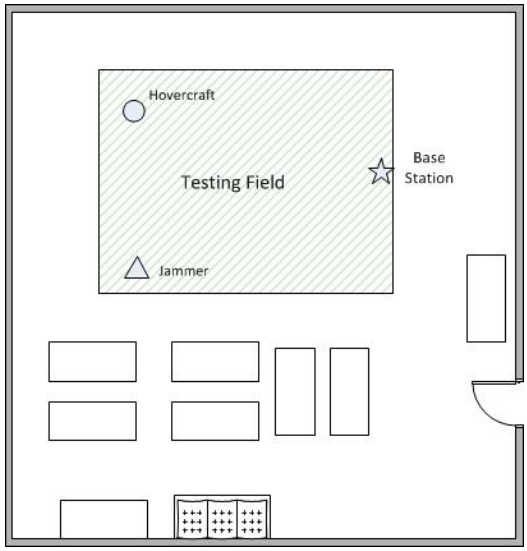

Figure 4.5: Test field

4.7 Analysis and Conclusion

4.7.1 Jamming Experiment Result

Five locations are chosen to place the jammer: four corners and a center point. With each jammer location, a graph is generated based on the status of the Hovercraft. Fig 4.10 has extra measurement points around the center. The intention of the plot is to reveal the pattern of the signal range and how the jammer interferes with the base station signal. But after the plot is generated (Fig 4.6 to Fig 4.10), there is no obvious pattern for this interference in 2D space. Jammed locations that are not affected are interleaved in the graph. Although test locations closed to jammer are more likely to be jammed, the result are not definitive.

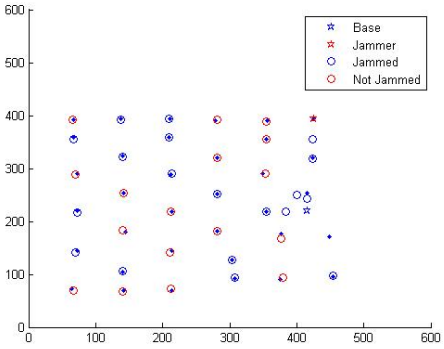

Figure 4.6: Jammer location: upper right corner

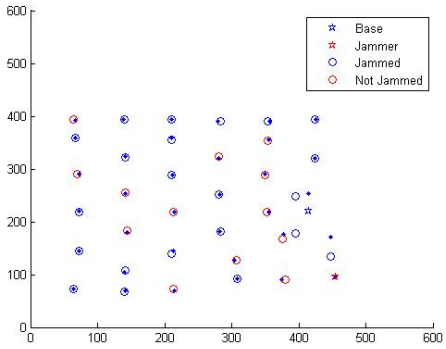

Figure 4.7: Jammer location: lower right corner

There are several constraints on this experiment settings. The RF modules used for testing have no built-in antenna other than a simple quarter-wavelength wire. The signal strength is not very stable across different directions. And the space is small, which is likely to cause much reflection from the walls around. Another possible issue is that the sample points might be too sparse.

4.7.2 Framework Usage

This application shows that the framework is capable of driving the DSP and providing basic inter-processor communication between ARM and DSP. But it is not the ideal application to reveal the advantage of the framework. An ideal application should have more computation on the DSP and less frequent IPC.

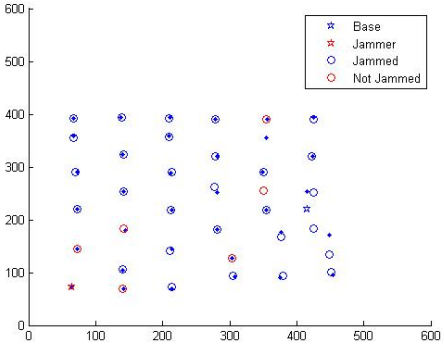

Figure 4.8: Jammer location: lower left corner

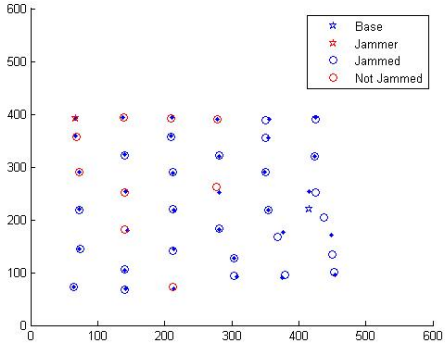

Figure 4.9: Jammer location: upper left corner

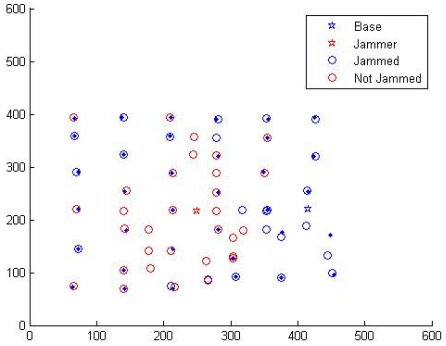

Figure 4.10: Jammer location: center

CHAPTER 5

FUTURE WORK

5.1 Framework Improvements

Since the framework is extremely flexible, it is better to provide a configuration script to the user to increase the ease of use and prevent changing the value of a macro in multiple files. Another essential improvement is to make the driver code thread-safe. Beyond the above idea, it is also possible to introduce a simple scheduler to run multiple DSP programs.

5.2 Wireless Jamming Experiment Improvements

To increase the overall robustness of the RF transmission and reception, a customized PCB board is favored over the breadboard. Also, better RF transceivers, such as those with built-in antennas, can reduce the number of components on the circuit board as well as increase the efficiency of the communication. And to model the propagation of electromagnetic waves generated by the RF transmitters, the Helmholtz equation is a good approach.

APPENDIX A

GUMSTIX DEVELOPMENT WIKI

A.1 Quick Start Guide

In order to use this framework, a working Gumstix Overo with Linux running on ARM is needed.

- The framework is tested based on Kernel version OMAP 3.5.
- Rootfs is from the same image as other Overo boards.
- No third party library is needed.

Step 1: Build and Load kernel module

1. Download code from repository.
2. Make (cross compile on host laptop).
3. Transfer the .ko file to the Gumstix.
4. Load MiniDSP module (Log in as root).

   ```
   rmmod dspdd.ko
   rm -f /dev/dspdd
   insmod dspdd.ko
   mknod /dev/dspdd c 248 0
   chmod 664 /dev/dspdd
   ```

   Note: The major number of the device might change on different machines.

5. Make will also cross compile a host program. The host program will run on ARM core to control DSP.

6. Transfer armhost executable to Gumstix.

Step 2: Prepare DSP program Executable

1. Download CCS from TI website DSP C code compiler tool chain is included in this IDE. Very similar to Eclipse.
2. Create a project and write the code.
3. Check out an existing example program.
4. Copy the boot.asm, dsplinker.cmd to the new project. Or just modify from an existing DSP projects.
5. Build project to get a .out DSP executable.
6. Download coff_parser source code.
7. Build it locally on host laptop.
8. Run coff_parser on the DSP executable(.out).
9. Transfer the generated bin file to Overo.

Step 3: Load and Run DSP program

1. Currently we should have
   DSP module loaded
   Arm host program executable
   DSP program bin file
2. Run ./armhost binfile_name
   Bin file will be loaded into DSP memory.
3. Release DSP to run the program.

A.2 Bring up

A.2.1 Build Kernel

Use Cross Compile to build Linux kernel on the host machine and then copy the image to the SD card. The following instructions are summarized from Gumstix website.

1. On the host machine, we need to get the cross-compiler. On Ubuntu, use the following command:

   ```
   sudo apt-get install gcc-arm-linux-gnueabi uboot-mkimage
   ```

2. Then we need to download the kernel source code on the host machine. I use version omap-3.5.

3. omap 3.5 can be configured to run on different hardware platforms, so we need a kernel configuration for our Gumstix Overo. The configuration I use is from Sakoman Gumstix configuration file.

4. Change the regulator_dummy = Y. And under BOARD section of the config file, boards other than Overo can be removed. The config file is attached in the end.

5. Give a name to the config file, like Overo_defconfig. The config file should be moved to the downloaded kernel working directory:

   ```
   arch/arm/configs/
   ```

6. Now run the configuration. Replace $< platform >$_defconfig with the name of the config file you just created.

   ```
   make ARCH=arm \\
   CROSS_COMPILE=arm-linux-gnueabi- <platform>_defconfig
   ```

7. Then configure the kernel to your needs:

   ```
   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- menuconfig
   ```

8. Then Compile the kernel

   ```
   make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- uImage -j4
   ```

   Replace the -j4 option with however many threads you want GNU make to utilize. Usually using twice as many threads as CPU cores available is good practice. When finished, the uImage kernel file can be found in the arch/arm/boot directory.

9. Kernel modules can be similarly compiled:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules -j4
```

10. Then you need to install the kernel modules to a preformatted microSD card (see next section) where the root file system is mounted at /media/rootfs. Run depmod on the gumstix to make sure these new modules can be found.

```
sudo INSTALL_MOD_PATH=/media/rootfs make ARCH=arm
CROSS_COMPILE=arm-linux-gnueabi- modules_install
```

A.2.2 Create a Bootable MicroSD card

1. Insert microSD to host machine.
   Use dmesg to determine the device filename of your microSD card in the form sdX, where X is a letter assigned by your computer.
   Example output from console:

```
[16033.652018] mmc0: new SDXC card at address aaaa
[16033.675007] mmcblk0: mmc0:aaaa SU64G 59.4 GiB
[16033.676463] mmcblk0: p1 p2
```

2. Unmount any mounted partitions on the attached device.

```
sudo umount /dev/mmcblk0{p1,p2}
```

3. Find the size of your card in the output of the following command:

```
sudo fdisk -l /dev/mmcblk0
```

   Example output from console:

```
Disk /dev/mmcblk0: 63.9 GB, 63864569856 bytes 255 heads,
63 sectors/track, 7764 cylinders,
total 124735488 sectors
Units = sectors of 1 * 512 = 512 bytes
```

```
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000
Device Boot Start End Blocks Id System
/dev/mmcblk0p1 * 63 144584 72261 c W95 FAT32 (LBA)
/dev/mmcblk0p2 144585 57625154 28740285 83 Linux
```

4. Divide the card size in bytes by 255 heads,63 sectors and 512 bytes per sector and round down to the nearest integer.
   This will give you the number of cylinders needed for your drive's geometry.
   Use 63864569856 as above

$$(63864569856) \div 255 \div 63 \div 512 = 7764$$

5. To use the uImage and root file system image, we need 2 partitions: a FAT partition containing the boot files and a Linux partition containing the root file system. To create these partitions:

   - Clean up any existing partition data:

     ```
     sudo dd if=/dev/zero of=/dev/mmcblk0 bs=1024 count=1024
     ```

   - Create the new partitions using sfdisk:

     ```
     sudo sfdisk -D -uM -H 255 -S 63 -C 7764 /dev/mmcblk0
     ```

     At the sfdisk prompt, type 0,64,0x0C,* followed by Enter to set up a 64 MB bootable FAT32 partition.
     At the following prompt, type 75,,,- followed by Enter to create the root file system partition.
     For the next two prompts, type 0,0,, followed by Enter. Make sure you do this twice.
     When prompted, type y + Enter to write the partition table.

6. Finally you must format the partitions. First, create the boot partition. Be sure to specify the number of the partition to format after /dev/mmcblk0 (e.g./dev/mmcblk0p1):

```
sudo mkfs.vfat -F 32 /dev/mmcblk01 -n boot
```

If your computer is missing the mkfs.vfat program, try installing the dosfsutils package and try again:

```
sudo apt-get install dosfsutils
```

Now create the Linux root filesystem partition:

```
sudo mke2fs -j -L rootfs /dev/mmcblk0p2
```

Your microSD card is now ready to be flashed with the boot and root file system images.

A.2.3 Back Up SD Card Image

If you want a backup that is used to restore your card (same type of SD card, same size). Using dd is easy. To generate an image file from your SD card:

```
dd if =/dev/mmcblk0{p1,p2} of= backup.img bs=4M
```

mmcblk0 is the device name under /dev . Use dmesg for the correct name. p1,p2 is the partition. I created separate image for each partition. bs is block size.
To restore your SD card with existing image:

```
dd if = backup.img of=/dev/mmcblk0{p1,p2} bs=4M
```

Warning: This sd card image is not equal to rootfs tar. If you dd the rootfs partition image, it will not work with your new kernel image. This is because dd img have more information than pure rootfs tar file. To create a rootfs from an existing SD card image please see next how-to.

A.2.4 Generate rootfs From Existing SD Card Image

1. You can use the following to compress the SDCard: Make sure you are under rootfs folder, where you see /bin,/lib, etc. You can create a script, but you need to modify a little bit to fit your settings.

```
#!/bin/bash
cd /media/rootfs/
tar -cpjvf /home/dir/compressed.tar.bz2 . --numeric-owner
```

2. You use the following to uncompress the file to the SDCard:

```
#!/bin/bash
tar -xpjvf /home/dir/compressed.tar.bz2
-C /media/rootfs/ --numeric-owner
```

Now you have a rootfs.tar.bz2 which is ready to be used.

A.3 Troubleshooting

1. Gumstix not boot up?
   System hangs. A red LED indicates that system is shut down. This does not happen when HDMI display is connected. The cause of this problem is that noise on the UART_RX3 line during boot can interrupt the normal booting sequence. Consider adding a pull-up resistor to a SYSEN-gated 1.8V supply on this line.

2. Boot stuck at Uncompressing Linux Image.....?
   Check Kernel Version, before 2.6.** (probably 2.6.34), serial console is no longer ttyS2, but ttyO2 (alphabet o, not zero). You should check U-boot environment parameter: console. Change it to ttyO2.

3. Linux boots successfully but stuck, no login available ?
   If you're interfacing with your Overo with the USB serial terminal, keep in mind that you won't see the *'login :'* prompt after booting Ubuntu unless you configure getty to listen on /dev/ttyS2. You'll need to do this while your MicroSD card is still mounted on host.

```
#cd /path/to/rootfs
#cat > etc/event.d/ttyS2 <<EOF
start on runlevel 2
start on runlevel 3
start on runlevel 4
```

```
start on runlevel 5
stop on runlevel 0
respawn exec
/sbin/getty 115200 ttyS2 EOF
```

You will also need to add ttyS2 to /etc/securetty to allow root logins there:

```
#echo "ttyS2" >> etc/securetty
```

Note: New kernel version should use ttyO2 in the above situation.

REFERENCES

[1] M. Yovits, *Advances In Computers*. Elsevier Science, 1993.

[2] *OMAP3530 Technical Reference Manual*, Texas Instruments, 2010.

[3] H. Doyu, "DSP gateway." [Online]. Available: http://sourceforge.net/projects/dspgateway/

[4] F. Contreras, "Beagleboard DSP clarification." [Online]. Available: http://elinux.org/BeagleBoard/DSP_Clarification

[5] Texas Instruments, "DSPBridge project." [Online]. Available: http://www.omappedia.org/wiki/DSPBridge_Project#About_DSP_Bridge

[6] Texas Instruments, "Programming guide for DSP/BIOS bridge." [Online]. Available: https://gforge.ti.com/gf/project/omapbridge/docman/Documents/

[7] C. Ring, "Syslink user guide." [Online]. Available: http://processors.wiki.ti.com/index.php/SysLink_UserGuide

[8] C. Ring, "C6EZRun." [Online]. Available: http://processors.wiki.ti.com/index.php/C6EZRun

[9] Texas Instruments, "Linux digital video software development kit (DVSDK) for OMAP3530/3525 digital media processors." [Online]. Available: http://www.ti.com/tool/linuxdvsdk-omap3530

[10] *TMS320C6000 Assembly Language Tools*, Texas Instruments, 2012.

[11] *Using OFD Utility to Create a DSP Boot Image*, Texas Instruments, 2004.

[12] *Creating a DSP Boot Image for Host Boot*, Texas Instruments, 2009.